Apellidos: **Nombre:** **Matrícula:**

Concurrencia (parte 1)/clave: a

Curso 2015–2016 - Convocatoria Extraordinaria (Julio 2016)
Grado en Ingeniería Informática / Grado en Matemáticas e Informática
UNIVERSIDAD POLITÉCNICA DE MADRID

NORMAS: Este es un cuestionario que consta de 7 preguntas. Todas las preguntas son de respuesta simple excepto la pregunta 7 que es una pregunta de desarrollo. La puntuación total del examen es de 10 puntos. La duración total es de una hora. El examen debe contestarse en las **mismas hojas**. No olvidéis rellenar **apellidos, nombre y número de matricula** en cada hoja.

Sólo hay una respuesta válida a cada pregunta de respuesta simple. Toda pregunta en que se marque más de una respuesta se considerará incorrectamente contestada. Toda pregunta incorrectamente contestada restará del examen una cantidad de puntos igual a la puntuación de la pregunta dividido por el número de alternativas ofrecidas en la misma.

Cuestionario

($1\frac{1}{2}$ puntos) 1. Suponiendo que hay tres procesos $t1$, $t2$ y $t3$, que cada uno ejecuta repetidamente las operaciones *uno*, *dos*, *tres* del CTAD que se especifica a continuación:

TIPO: MiCTAD $= (a : \mathbb{B} \times b : \mathbb{B})$
INICIAL: $self = (\text{Falso}, \text{Falso})$
INVARIANTE: $\neg self.a \vee \neg self.b$

CPRE: $\neg self.a$
uno()
POST: $self^{pre} = (a, b) \wedge self = (a, \neg b)$
CPRE: $self.a \vee self.b$
dos()
POST: $self^{pre} = (a, b) \wedge self = (\neg a, \neg b)$
CPRE: Cierto
tres()
POST: $self^{pre} = (a, b) \wedge self = (\text{Cierto}, b)$

Se pide marcar la afirmación correcta:

(a) En el programa no podría llegar a violarse la invariante

(b) En el programa podría llegar a violarse la invariante.

(Sugerencia: Dibuja aquí el grafo de los estados por los que puede pasar el recurso.)

(1 punto) 2. Dadas las siguientes clases `MiHilo` y `Dato`:

```
class MiHilo extends Thread {
    private volatile Dato d;

    public class MiHilo(Dato d) {
        this.d = d;
    }
    public void run () {
        d.m();
    }
}
```

```
class Dato {
    private int n;

    public class Dato () {
        n = 0;
    }
    public void m () {
        n++;
    }
}
```

Se pide marcar la afirmación correcta:

(a) La ejecucíón del método `m()` puede ser una sección crítica

(b) La ejecucíón del método `m()` nunca será una sección crítica

(1 punto) 3. Dado el siguiente `main` ¿cuántos hilos *como máximo* habría simultáneamente ejecutando en el programa?:

```
public static void main(String[] args) {
  MiHilo t1 = new MiHilo (new Dato());
  t1.start();
  t1.run();
}
```

Se pide marcar la afirmación correcta:

(a) 1 hilo.

(b) 2 hilos.

(c) 3 hilos.

(1½ puntos) 4. Dado un programa concurrente en la que dos *threads* instancias de las clases `A` y `B`, `C` comparten una variable `n`:

```
static int n = 0;
static Semaphore s1 = new Semaphore(1);
static Semaphore s2 = new Semaphore(0);
```

```
static class A
  extends Thread {
    public void run() {
        s2.await();
        n = 3 * n;
        s1.signal();
    }
}
```

```
static class B
  extends Thread {
    public void run() {
        s1.await();
        n = n + 2;
        s2.signal();
    }
}
```

```
static class C
  extends Thread {
    public void run() {
        s1.await();
        n = n + 4;
        s2.signal();
    }
}
```

¿Cuáles son los posibles valores de `n` tras **terminar** los tres threads?

(a) 6, 10, 14 y 18

(b) 14 y 6

(c) 10 y 14

(1 punto) 5. Si en el código anterior el semáforo `s1` se inicializa a 2:

(a) No está garantizada la terminación de las tres tareas.

(b) No está garantizada la exclusión mutua en el acceso a `n`.

Apellidos: **Nombre:** **Matrícula:**

(1 punto) 6. Dado el siguiente código que trata de implementar exclusión mutua en los accesos a la variable compartida `n`:

```
static class MutexEA {
   static final int N_PASOS = 1000000;

   // Variable compartida
   volatile static int n = 0;

   // Variables para asegurar mutex
   volatile static boolean en_sc_inc =
      false;
   volatile static boolean en_sc_dec =
      false;

   static class Incrementador
      extends Thread {
      public void run () {
         for (int i = 0;
              i < N_PASOS;
              i++) {
            en_sc_inc = true;
            while (en_sc_dec) {
               en_sc_inc = false;
               en_sc_inc = true;
            }
            n++;
            en_sc_inc = false;
         }
      }
   }

   static class Decrementador
      extends Thread {
      public void run () {
         for (int i = 0;
              i < N_PASOS;
              i++) {
            en_sc_dec = true;
            while (en_sc_inc) {
               en_sc_dec = false;
               en_sc_dec = true;
            }
            n--;
            en_sc_dec = false;
         }
      }
   }

   public static final void
      main(final String[] args)
      throws InterruptedException {
      Thread t1 = new Incrementador();
      Thread t2 = new Decrementador();

      t1.start();
      t2.start();

      t1.join();
      t2.join();
   }
}
```

Se pide marcar la afirmación correcta:

(a) El programa no garantiza la exclusión mutua

(b) El programa puede acabar en un interbloqueo.

(c) El programa no cumple la propiedad de ausencia de inanición

(3 puntos) 7. **Se pide** especificar un recurso compartido que gestiona un sistema de backups que dispone de varios discos duros para realizar las copias de seguridad. En este recurso interactúan dos tipos de procesos: unos que se encargan de *hacer backups* y otros que se encargan de ir *reponiendo los discos* a medida que éstos se van llenando.

El sistema de backups dispone de *NUM_DISCOS* discos duros con capacidad *MAX_BYTES*. La operación *hacerBackup*(*size*) solicita la realización de un backup de tamaño *size* bytes. El valor del parámetro *size* siempre debe ser menor que *MAX_BYTES*. El sistema calcula en qué disco hacer el backup mediante la operación *size* $\%$*NUM_DISCOS*. Por ejemplo, si *size* $= 14$ y *NUM_DISCOS* $= 5$ el sistema hará el backup en el disco número 4. En caso de se intente realizar un backup y el disco asignado ya haya superado su capacidad, el proceso deberá bloquearse hasta que vuelva a haber espacio en el disco correspondiente. La operación *reponerDisco*(i) deberá bloquearse mientras el disco *i* no esté lleno. Cuando el disco se haya llenado, es decir, haya superado el tamaño *MAX_BYTES*, *reponerDisco*(i) deberá reponer el disco *i* indicando que el espacio ocupado del disco *i* es de 0 bytes. Se ha definido el tipo *PosDisco* para indicar que *i* siempre estará entre los valores válidos, es decir, $[0..$*NUM_DISCOS* $- 1]$. Se asume, por simplificar, que se puede superar el tamaño máximo del disco en la última operación, es decir, si un disco de tamaño 100, tiene ocupados 80 y le llega un backup de 30, este podrá hacerse aunque se supere la capacidad del disco.

C-TAD SistemaBackup

OPERACIONES

ACCIÓN hacerBackup: $\mathbb{Z}$*[e]*
ACCIÓN reponerDisco: *PosDisco[e]*

SEMÁNTICA
DOMINIO:

TIPO: *SistemaBackup* =

TIPO: *PosDisco* $= 0..$*NUM_DISCOS* $- 1$

INICIAL:

PRE:

CPRE:

hacerBackup(size)

POST:

CPRE:

reponerDisco(i)

POST: